**Q1.**

|  |
| --- |
| Pros of Inheritance in C++ |
| 1. Code Reusability |
| Inheritance allows you to reuse existing classes by inheriting their attributes and behaviors, reducing the need to rewrite code. |
| 2. Hierarchical Modeling |
| You can create a hierarchical structure of classes, which can represent real-world relationships and model complex systems effectively. |
| 3. Polymorphism |
| Inheritance is a fundamental concept for achieving polymorphism in C++. It enables you to create objects of derived classes and use them interchangeably with base class objects. |
| 4. Encapsulation |
| Derived classes can inherit the members (methods and attributes) of base classes, allowing you to encapsulate common functionality and extend it as needed. |
| 5. Code Extensibility |
| You can add new features to existing classes by creating derived classes, without modifying the base class code, which helps maintain and extend the software easily. |

|  |  |
| --- | --- |
| Cons of Inheritance | Description |
| Tight Coupling | Inheritance creates a tight coupling between the base and derived classes. Changes in the base class may impact all derived classes, making the code harder to maintain. |
| Fragile Base Class Problem | Modifying the base class can unintentionally affect derived classes, potentially introducing bugs. This is known as the "fragile base class" problem. |
| Lack of Encapsulation | Inheritance can expose the internal details of a base class to derived classes, which can lead to violations of encapsulation and hinder code maintainability. |
| Limited Multiple Inheritance | C++ supports multiple inheritance, but it can lead to ambiguity and complexity when a class inherits from multiple base classes. This can result in the "diamond problem." |
| Inflexible Hierarchy | Inheritance hierarchies can be challenging to change once established. Adding new classes or modifying the hierarchy can be cumbersome and lead to code bloat. |
| Reduced Code Reusability | Inheritance may not always be the best mechanism for code reuse, as it can introduce unnecessary dependencies between classes. Composition can offer more flexible and reusable solutions. |
| Difficulty in Testing | Testing derived classes can be challenging, as they inherit the behavior of the base class. This can make it difficult to isolate and test specific functionality. |

**Q2.**

**HYBRID INHERITANCE**

#include <iostream>

using namespace std;

class *player*{

    protected:

        char name[20];

        char gender;

        int age;

};

class *physique* : public *player*{

    protected:

        float height;

        float weight;

};

class *location*{

    protected:

        char city[20];

        int pin;

};

class *game* : public *physique*, *location*{

    public:

        char game[20];

        void get(){

            cout<<"Name: ";cin>>name;

            cout<<"Gender: ";cin>>gender;

            cout<<"Age: ";cin>>age;

            cout<<"Height: ";cin>>height;

            cout<<"Weight: ";cin>>weight;

            cout<<"City: ";cin>>city;

            cout<<"Pin: ";cin>>pin;

            cout<<"Game: ";cin>>game;

        }

        void output(){

            cout << name << endl;

            cout << gender << endl;

            cout << age << endl;

            cout << height << endl;

            cout << weight << endl;

            cout << city << endl;

            cout << pin << endl;

            cout << game << endl;

        }

};

int main(){

*game* G;

    G.get();

    G.output();

    return 0;

}

**Multipath Inheritance**

#include <iostream>

using namespace std;

class *A*{

    protected:

        int a;

};

class *B* : virtual public *A*{

    protected:

        int b;

};

class *C* : virtual public *A*{

    protected:

        int c;

};

class *D* : public *B*,*C*{

    int d;

    public:

        D(){

            cout << "Enter : a b c d\n";

            cin>>a>>b>>c>>d;

            cout<<a<<" "<<b<<" "<<c<<" "<<d;

        }

};

int main(){

*D* User;

    return 0;

}

**Q3.**

A common constructor is typically the default constructor, which is provided by the compiler if you don't define any constructors for your class. The default constructor initializes the object's data members to some default values.

#include <iostream>

using namespace std;

class MyClass {

    public:

        MyClass() {

            std::cout << "Default constructor invoked." << endl;

        }

};

int main() {

    MyClass obj;

    return 0;

}

**Q4.**

A wild pointer is a pointer that does not point to a valid memory location. This can occur when a pointer is either uninitialized or when it has been freed or deleted, but the program continues to use it. Wild pointers are a common source of runtime errors and can lead to unpredictable behavior in a C++ program, including crashes and data corruption.

#include <iostream>

using namespace std;

int main() {

    int\* wildPointer; *// This is a wild pointer - uninitialized and points to an undefined location.*

*// Attempting to access or modify the value through the wild pointer will lead to undefined behavior.*

    \*wildPointer = 42; *// This is a dangerous operation.*

    cout << "Value through wild pointer: " << \*wildPointer << endl; *// May result in a crash or garbage value.*

    return 0;

}

**Q5.**

#include <iostream>

using namespace std;

int main() {

    int intValue = 42;

    double doubleValue = 3.14;

    char charValue = 'A';

    void\* genericPointer; *// Declaring a void pointer*

*// Assigning the address of an int variable to the void pointer*

    genericPointer = &intValue;

*// To use the value pointed to by the void pointer, you need to cast it back to the appropriate type*

    int\* intPointer = (int \*)genericPointer;

    cout << "Value through int pointer: " << \*intPointer << endl;

*// Assigning the address of a double variable to the void pointer*

    genericPointer = &doubleValue;

*// Casting back to the double type to access the value*

    double\* doublePointer = (double \*)genericPointer;

    cout << "Value through double pointer: " << \*doublePointer << endl;

*// Assigning the address of a char variable to the void pointer*

    genericPointer = &charValue;

*// Casting back to the char type to access the value*

    char\* charPointer = (char \*)genericPointer;

    cout << "Value through char pointer: " << \*charPointer << endl;

    return 0;

}

Q6.

**#include <iostream>**

**using namespace std;**

**class *A*{**

**public:**

**A(){**

**cout << "Base Class A" << endl;**

**}**

**};**

**class *B* : public *A*{**

**public:**

***A* A1;**

**B(){**

**cout << "Derived Class B" << endl;**

**}**

**};**

**int main(){**

***B* User;**

**return 0;**

**}**